DOCUMENT RESUME

ED 057 581 EM 009 421

AUTHOR Lukas, George; And Others

TITLE LOGO Teaching Sequences on Strategy in Problem-Solving and Story Problems in Algebra. Teacher's Text and Problems.

INSTITUTION Bolt, Beranek and Newman, Inc., Cambridge, Mass.
SPONS AGENCY National Science Foundation, Washington, D.C.
REPORT NO R-2165
PUB DATE 30 Jun 71
NOTE 226p.; Programming-Language as a Conceptual Framework for Teaching Mathematics, Volume Three; See also EM009 419, EM 009 420, EM 009 422

EDRS PRICE MF-$0.65 HC-$9.87
DESCRIPTORS *Computer Assisted Instruction; *Computer Programs; *Mathematics Instruction; *Problem Solving; Secondary School Mathematics
IDENTIFIERS Project LOGO

ABSTRACT

In order to provide high school students with general problem-solving skills, two LOGO computer-assisted instruction units were developed--one on the methods and strategies for solution and a second on the relation between formal and informal representations of problems. In both cases specific problem contexts were used to give definition and articulation to central notions like problem, problem form, solution method, and optimal strategy. The unit on strategies in problem solving illustrates strategy formation in two contexts--extrapolating number sequences and exploring mazes. The unit on story problems in algebra attempts to help students learn to convert a story problem into formal mathematical terms. For more information about the LOGO project, see volumes I, II, and IV of the report (EM 009 419, EM 009 420, and EM 009 422). (JY)

BOLT BERANEK AND NEWMAN INC

CONSULTING • DEVELOPMENT • RESEARCH

Report No. 21[illegible]5

Volume 3

PROGRAMMING-LANGUAGES AS A CONCEPTUAL FRAMEWORK FOR TEACHING MATHEMATICS

LOGO Teaching Sequences on

Strategy in Problem-Solving

and

Story Problems in Algebra

Submitted to:

National Science Foundation
Office of Computing Activities
1800 G Street, N.W.
Washington, D. C. 20550

Contract NSF-C 615

30 June 1971

Report No. 2165, Volume 3

Strategy in Problem-Solving
and
Story Problems in Algebra

FOREWORD

High-school students seldom acquire *general* problem-solving skills. They lack a meta-language for discussing problem-solving and they lack suitable models where the methods of problem-solving are brought to the fore. The two LOGO units in this volume deal directly with these issues. In the first unit, the focus is on methods and strategies for solution; in the second, on the relation between formal and informal representations of problems. In both cases specific problem contexts are used to give definiteness and articulation to central notions like problem, problem form, solution method, and optimal strategy.

In the unit "Strategy in Problem-Solving," we develop extended sequences illustrating strategy formation in two rather different contexts -- extrapolating number sequences and exploring mazes. These examples are designed to give a close-up view into problem-solving work, while showing the flavour of different approaches, and indicating the scope of results possible. Substantive mathematical issues arise naturally along the way and are treated from an intuitive, constructive point of view. We chose our examples to facilitate such treatment -- many other good choices can be made, including studying abstract structures such as groups, games of strategy, and formal manipulation methods.

The unit "Story Problems in Algebra" deals with a central and difficult issue to beginning students. All too often, a student does not realize that the problem of converting a story problem into a problem into formal mathematical terms is a problem in translation. Although the translation rules cannot be made as precise

as those for translation between formal systems, they are considerably more precise than those governing the translation say, between English and Chinese. The student who does realize that he is faced with a translation problem still has the problem of formulating the rules -- they are never given to him. In this unit we shall try to give him such "rule-formulating" abilities.

We start with a relatively formal translation problem -- the translation from infix to prefix notation. This serves to shed considerable light on the operations needed to carry forward the less-formal translation and seems a great deal easier. Next we deal with story problems themselves, but instead of taking the usual route of translating from story problems into equations, we translate in the other direction. This route is considerably easier and focuses attention on the translation process itself. For these purposes we develop two kinds of story problem translations -- prefix to story and infix to story. The former is used with a variety of different story problem types involving a single unknown; the latter with problems involving a system of equations.

Volume 3, Part 1

STRATEGY IN PROBLEM-SOLVING

Teacher's Text

and

Problems

The LOGO Project

NSF-C 615

George Lukas
Philip Faflick
Wallace Feurzeig
Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Mass. 02138

CONTENTS

LOGO UNIT ON STRATEGY IN PROBLEM-SOLVING

0. Introduction

This unit develops the art of solving problems by carrying through some examples. We have chosen problem situations that are amenable to different kinds and levels of attack, and that can be approached initially by elementary, intuitively-grasped methods such as guessing and searching. These basic methods are naturally expressed as LOGO procedures. Using these, problem-solving strategies are developed as higher-level procedures either by extending the basic ones or by transforming the problems themselves.

We have sought throughout the LOGO teaching units to counter the harmful impression often given to students that "right" or "best" theorems, algorithms, or solutions are born whole, or are obvious, trivial constructions (or even always exist). In this unit we do not produce optional strategies as instant solutions, like rabbits out of a hat. Instead, the concepts of "good" and "best" strategies arise gradually along with the development of criteria for comparing strategies. Primary emphasis is given to the process of building, testing, and improving strategies.

The unit is divided into three parts. We begin studying strategies for a simple game -- guessing an unknown number. Random and systematic guessing and searching methods are easily introduced in this context.

We next extend number-guessing to a richer problem-solving situation, extrapolating number sequences. Here there is enough additional structure to permit developing a variety of different strategies, some building on others and some essentially

independent of others. Some strategies have to do with transforming the sequences to be extrapolated, and others with modifying the rules (LOGO procedures) for generating solutions. The development is carried forward in a "learning" framework. The capabilities of the extrapolation program are continually improved by the incorporation of new strategies. Issues of program organization such as the order of application of strategies, arise naturally.

As a last example, we study an apparently very different situation -- finding effective methods of traversing mazes. The context here is geometric and topological rather than numerical. The basic procedures for moving to adjacent cells are developed first. Using these, various systematic strategies such as "always bear to the left" are defined. Random strategies (and substrategies) are introduced to try to avoid endless repetitive loops and "dead-ends". The possibilities for developing more complex and interesting strategies are further enhanced by marking cells as they are traversed, thus enabling the use of history-dependent strategies.

The LOGO treatment of maze strategies was designed by George Lukas with major contributions from Philip Faflick. George Lukas and Wallace Feurzeig collaborated in the original design of the LOGO number sequence extrapolation material. Philip Faflick contributed to the realization and did the major writing of both this and the maze material. Wallace Feurzeig wrote the introductory section on number-guessing strategies. Pearl Stockwell provided valuable editorial assistance, technical as well as clerical, in the course of transcribing this unit to typed form.

1. NUMBER GUESSING

There are many games of the kind where one person attempts to guess another's "secret" object, whether it is "in this room" or "out of this world", or neither. Among these, Twenty Questions is probably the best known. Some of the number-guessing games are especially well-suited for developing efficient guessing strategies, starting from the simplest guessing procedures. Perhaps the least-sophisticated such game is number guessing without responsive feedback. The situation is: one person thinks of a number, say between Ø and 999, and another tries to guess the number; each time a guess is made the guesser is informed whether or not his guess is correct.

In this form of guessing game no further information about the guess is given -- not even hints like "you're getting warmer" or "you're ice cold". So it's not immediately obvious how we should proceed in making good guesses. Let's start very systematically, by counting up from Ø until we get to the secret number. This is a tedious process, so we'll write a LOGO procedure COUNT-UP to do it for us.

```
TO COUNT-UP /N/                          (/N/ is the current guess)
1Ø PRINT SENTENCE OF SENTENCE OF
   "IS YOUR NUMBER" /N/ "?"              (Announce the guess)
2Ø MAKE
    NAME:  "ANSWER"                      (Find whether or not the
    THING:  REQUEST                       guess is correct)
3Ø TEST IS /ANSWER/ "YES"
4Ø IF FALSE COUNT-UP                     (If not, increase guess and
    (SUM OF /N/ AND 1)                    try again)
5Ø IF TRUE PRINT "HOORAY--I GOT IT!"
END
```

Let's use COUNT-UP to guess a number.

```
←COUNT-UP Ø                          (COUNT-UP starts by guessing Ø)
IS YOUR NUMBER Ø?
*NO
IS YOUR NUMBER 1?
*NO
IS YOUR NUMBER 2?
*NO
 : : :
IS YOUR NUMBER 333?
*YES                                 (Assuming 333 is the answer)
HOORAY--I GOT IT!
←
```

COUNT-UP takes a long time to get to large numbers. Similarly, the opposite kind of counting procedure, COUNT-DOWN, (for counting backwards from the biggest number) is slow to arrive at small numbers. A more effective guessing procedure should somehow be less biased in favor of high or low numbers. We would like a guessing procedure which better mixes successive guesses among large and small numbers. At the same time, we don't want it to a priori favor evens or primes or, generally, numbers with other special properties. A good way of making such unbiased guesses is by a random or "blind" process in which each possible number in the guessing interval is equally likely to be chosen and where successive choices are independent or uncorrelated. It's easy to write a LOGO procedure BLIND-GUESS to do this kind of guessing.

```
TO BLIND-GUESS
1Ø MAKE
    NAME:  "GUESS"                   (Make a 3-digit random number
    THING:  RANDOM-NUMBER 3           for use as the guess)
2Ø PRINT SENTENCE OF SENTENCE OF
   "IS YOUR NUMBER" /GUESS/ "?"      (Announce the guess)
3Ø MAKE "ANSWER" REQUEST             (Find whether or not the guess
                                      is correct)
4Ø TEST IS /ANSWER/ "YES"
5Ø IF FALSE BLIND-GUESS              (If not, guess again)
6Ø IF TRUE PRINT "HOORAY--I GOT IT!"
END
```

Here is an illustration of the operation of BLIND-GUESS.

```
←BLIND-GUESS
IS YOUR NUMBER 17?
*NO
IS YOUR NUMBER 694?
*NO
IS YOUR NUMBER 17?
*NO, AS I ALREADY SAID!
  : : :
IS YOUR NUMBER 333?
*YES
HOORAY--I GOT IT!
←
```

BLIND-GUESS uses the subprocedure RANDOM-NUMBER /L/ which outputs a random number of /L/ digits length.

```
TO RANDOM-NUMBER /L/
1Ø TEST IS /L/ Ø                        (If no more digits are needed)
2Ø IF TRUE OUTPUT /EMPTY/               (Terminate process)
3Ø OUTPUT WORD OF                       (Otherwise, attack another
     RANDOM                              random digit and repeat
     RANDOM-NUMBER (DIFF /L/ AND 1) /L/-1 more times)
END
```

Thus:

```
←PRINT RANDOM-NUMBER 2
49
←PRINT RANDOM-NUMBER 1Ø
Ø924366Ø15
```

An evident way of improving BLIND-GUESS is to insure that it generates a new guess each time, by remembering previous guesses and disallowing repetitions of them. But, no further means of reducing guesses is possible. In this very limited kind of number-guessing situation, the information gained on any incorrect guess is not sufficient to eliminate any other number in the domain.

1.1 Directing Successive Guesses

A genuine improvement in the game comes about by giving the guesser one more piece of information about his guess. Instead of merely responding with a "no" to an incorrect guess, as before, we will also say whether the guess is too high or too low. In this somewhat extended game, the strategy of blind guessing is obviously seen to be unintelligent. If the guesser is told that his guess is too high, he clearly should choose a lower number for his next guess. Similarly, a too low guess should cause the guesser to make a bigger next guess. Let's start by writing such a procedure, TRAP, that operates as follows. Since we have to get this process started, and we don't have any feedback yet, let's begin with a randomly chosen guess. If a guess is too big, TRAP will decrement it by one, if too small TRAP will add one to it.

```
TO TRAP /GUESS/                    (/GUESS/ is the starting guess)
1Ø PRINT SENTENCE OF
    "MY GUESS IS" /GUESS/
2Ø PRINT "TELL ME--IS THAT HIGH, LOW, OR OK?"
3Ø MAKE "ANSWER" REQUEST
4Ø TEST IS /ANSWER/ "HIGH"
5Ø IF TRUE TRAP (DIFFERENCE OF /GUESS/ AND 1)
6Ø TEST IS /ANSWER/ "LOW"
7Ø IF TRUE TRAP (SUM OF /GUESS/ AND 1)
8Ø TEST IS /ANSWER/ "OK"
9Ø IF TRUE PRINT "HOORAY--I GOT IT!"
1ØØ IF FALSE TRAP /GUESS/          (Try again if /ANSWER/ is not
END                                "HIGH", "LOW", or "OK")
```

Here is an example of the operation of TRAP. Let's start it with a 3-digit random number, say 422, generated by RANDOM-NUMBER 3. If our secret number happened to be 131, this is what would happen:

```
←TRAP (RANDOM-NUMBER 3)
MY GUESS IS 422
TELL ME--IS THAT HIGH, LOW, OR OK?
*HIGH
MY GUESS IS 421
TELL ME--IS THAT HIGH, LOW, OR OK?
*HIGH
MY GUESS IS 42Ø
TELL ME--IS THAT HIGH, LOW, OR OK?
  : : :
MY GUESS IS 131
TELL ME--IS THAT HIGH, LOW, OR OK?
*OK
HOORAY--I GOT IT!
←
```

So, TRAP leads to lengthy (and dull) exchanges. The difficulty with this procedure is that it is too conservative. The only surprise in its operation is at the start when it finds, from its first (random) guess, an upper or lower bound. From that point on, it closes in on the answer inevitably, but very very slowly. It cannot overshoot (or undershoot) but it can take an enormous number of guesses.

A more efficient strategy comes about from making the guess depend upon both an upper bound and a lower bound. If we know that the answer lies between two numbers, we certainly should choose an intermediate number for our guess. If that number turns out to be too high (too low), it defines a new upper (lower) bound and thus we have further reduced the range of numbers for making our next guess. So this kind of procedure, like TRAP, will converge, though it may make many underestimates and overestimates along the way. And, hopefully, it will be a good deal more efficient on the average.

Let's write such a procedure SQUEEZE to reduce the interval of possible answers at each stage of guessing. The interval is

bounded on the low side by /BOTTOM/ and on the high side by /TOP/. SQUEEZE uses a subprocedure INSIDE for randomly choosing as its next guess a number within the specified interval.

```
TO SQUEEZE /BOTTOM/ /TOP/
1Ø MAKE
    NAME:  "GUESS"
    THING:  INSIDE OF /BOTTOM/ AND /TOP/
2Ø PRINT SENTENCE OF
    "MY GUESS IS" /GUESS/
3Ø PRINT "AM I HIGH, LOW, OR OK?"
4Ø MAKE "ANSWER" REQUEST
5Ø TEST IS /ANSWER/ "HIGH"          (If guess is too large)
6Ø IF TRUE SQUEEZE /BOTTOM/         (Try again using /GUESS/-1 as a
    (DIFF OF /GUESS/ AND 1)          better upper bound)
7Ø TEST IS /ANSWER/ "LOW"           (If guess is too small)
8Ø IF TRUE SQUEEZE (SUM OF          (Try again using /GUESS/+1 as a
    /GUESS/ AND 1) /TOP/             better lower bound)
9Ø TEST IS /ANSWER/ "OK"
1ØØ IF TRUE PRINT "HOORAY--
     I GOT IT!"
11Ø IF FALSE SQUEEZE /BOTTOM/ /TOP/
END
```

The subprocedure INSIDE uses the procedure RANDOM-NUMBER /L/, previously defined, to generate a random increment for /BOTTOM/.

```
TO INSIDE /BOTTOM/ /TOP/
1Ø MAKE "GAP" (DIFFERENCE OF /TOP/ AND /BOTTOM/)
2Ø MAKE "R" RANDOM-NUMBER OF (COUNT OF /GAP/)
3Ø TEST GREATERP /GAP/ /R/          (If interval length is greater
                                     than random increment)
4Ø IF TRUE OUTPUT                   (Use it)
    SUM OF /BOTTOM/ AND /R/
5Ø GO TO LINE 2Ø                    (Otherwise, try again)
END
```

Thus,

```
←PRINT INSIDE OF 35 AND 144
47
←
```

Let's try SQUEEZE in the same example we used above with TRAP.

```
←SQUEEZE Ø 999
MY GUESS IS 422
AM I HIGH, LOW, OR OK?
*HIGH
MY GUESS IS 375
AM I HIGH, LOW, OR OK?
*HIGH
MY GUESS IS 293
AM I HIGH, LOW, OR OK?
*HIGH
MY GUESS IS 144
AM I HIGH, LOW, OR OK?
*HIGH
MY GUESS IS 48
AM I HIGH, LOW, OR OK?
*LOW
MY GUESS IS 77
AM I HIGH, LOW, OR OK?
*LOW
MY GUESS IS 139
AM I HIGH, LOW, OR OK?
*HIGH
MY GUESS IS 118
AM I HIGH, LOW, OR OK?
*LOW
MY GUESS IS 136
AM I HIGH, LOW, OR OK?
*HIGH
MY GUESS IS 124
AM I HIGH, LOW, OR OK?
*LOW
MY GUESS IS 129
AM I HIGH, LOW, OR OK?
*LOW
MY GUESS IS 131
AM I HIGH, LOW, OR OK?
*OK
HOORAY--I GOT IT!
←
```

1.2 Binary Search

SQUEEZE shows the kind of improved convergence pattern that we expected relative to TRAP. We can speed up the operation of

SQUEEZE further by noting that there is a great deal of buffeting possible when the sequence of random choices made by INSIDE clusters on one or the other side of the interval, causing it to contract rather slowly even when the gap size has become quite small.

Therefore, we will replace the random incrementation procedure used in INSIDE by a procedure which splits the uncertainty evenly between the lower bound and the upper bound. We'll write a new procedure MIDDLE to directly output the mid-point of the gap as a best guess. Let's first replace line 1Ø of SQUEEZE by:

```
1Ø MAKE
    NAME:  "GUESS"
    THING:  MIDDLE OF /BOTTOM/ AND /TOP/
```

MIDDLE is, simply:

```
TO MIDDLE /BOTTOM/ /TOP/
1Ø OUTPUT QUOTIENT OF (SUM OF /BOTTOM/
    AND /TOP/) AND 2
END
```

Thus,

```
←PRINT MIDDLE OF 2 98
5Ø
←PRINT MIDDLE OF 175 1ØØØ
587
←PRINT MIDDLE OF 44 45
44
```

Using MIDDLE and SQUEEZE, let's replay the example used above.

```
←SQUEEZE Ø 999
MY GUESS IS 499
AM I HIGH, LOW, OR OK?
*HIGH
MY GUESS IS 249
AM I HIGH, LOW, OR OK?
*HIGH
```

```
MY GUESS IS 124
AM I HIGH, LOW, OR OK?
*LOW
MY GUESS IS 186
AM I HIGH, LOW, OR OK?
*HIGH
MY GUESS IS 155
AM I HIGH, LOW, OR OK?
*HIGH
MY GUESS IS 139
AM I HIGH, LOW, OR OK?
*HIGH
MY GUESS IS 131
AM I HIGH, LOW, OR OK?
*OK
HOORAY--I GOT IT!
←
```

The strategy of splitting the interval of search in half on each round is called binary search. In the example shown, it required fewer guesses than the previously used strategies. Let's see whether or not that was an accidental result of our choice of the interval size or the number to be guessed. (One might observe, for example, that if the unknown number had been 134 instead of 131, binary search would have taken 10 guesses instead of 7.)

Assume we are guessing in the interval (A,B) and that N the number to be guessed is somewhere in the interval, i.e., $A \leq N \leq B$. Using TRAP we only eliminate one possibility with each guess. Using binary search our guess, (B-A)/2, eliminates half the possibilities -- the subsequent guessing interval is either (A,(B-A)/2) or ((B-A)/2,B).

Using SQUEEZE, and randomly guessing between A and B, eliminates only one-third of the possibilities -- the argument is as follows: Let's call our guess G, where $A \leq G \leq B$. Assume first that G is too high. If so, TRAP eliminates B-G possibilities.

The probability that G is too high is simply (G-A)/(B-A) (i.e., the relative frequency of high guesses). Thus, the number of possibilities eliminated by a too high choice is

$$\frac{(G-A)}{(B-A)}\ (B-G).$$

Similarly, when G is too low it eliminates G-A possibilities. And this is true with probability (B-G)/(B-A). Thus, the number of possibilities eliminated by a too low choice is

$$\frac{(B-G)}{(B-A)}\ (G-A).$$

So altogether, adding these, we have

$$\frac{(G-A)}{(B-A)}\ (B-G)+\frac{(B-G)}{(B-A)}\ (G-A) = 2\frac{(G-A)(B-G)}{(B-A)}$$

possibilities eliminated on the average. Integrating over all possible guesses within the interval, we get

$$\frac{2}{(B-A)}\int_{A}^{B} (G-A)(B-G)dG = (B-A)/3$$

Finally then, SQUEEZE with random guessing eliminates one-third of the possibilities and hence is less efficient than binary search.

Binary search is clearly a better strategy than the others we have considered. We might ask if there is a *best* strategy for guessing numbers. And we might investigate other strategies that might appear better than binary search. For example, if it is good to divide the interval of search into two parts, might it not be better to divide it into three or more parts instead?

Just as we calculated the effect of binary search in reducing remaining choices, we can calculate it for the case where the guess divides the interval into three parts (ternary search)

instead. The probability of a too low guess here is 1/3 and the fraction of cases eliminated thereby is 2/3; the probability of a high guess is 2/3 and the fraction of cases eliminated is 1/3. Thus, at each stage 1/3 x 2/3 + 2/3 x 1/3, or 4/9, of the possibilities are eliminated and this is somewhat poorer than the fifty percent reduction obtained with binary search. Similarly, the n-ary search efficiency is

$$\frac{1}{n} \cdot \frac{n-1}{n} + \frac{n-1}{n} \cdot \frac{1}{n} = \frac{2(n-1)}{n^2} \quad ,$$

so the efficiency gets worse with increasing n.

Instead of studying number-guessing strategies further, we can extend the guessing task somewhat and thereby obtain a richer situation that calls for new kinds of strategies. We want to consider guessing problems with more structure than can be provided by guessing a single isolated number. Instead, we will provide more initial information by giving a *sequence* of numbers satisfying some (unknown) generation rule. The task will be to guess the rule or, rather, to guess the next element of the sequence by finding some generation rule that correctly enumerates the numbers in the sequence thus far.

In this problem situation we can develop, not only more varied strategies, but also a system which applies given strategies, and others derived from them, more and more effectively to new problems. We can build a program which in some very definite sense grows and "learns" as it is used.

2. NUMBER-SEQUENCE EXTRAPOLATION

2.1 Sequences and the Extrapolation Problem

We shall be concerned next with another kind of guessing game -- guessing the next number in a sequence of numbers. An illustrativ problem is:

given the sequence Ø 1 2 3 4 5
what is the next number?

This kind of problem is not mathematically well-defined because, of course, there are an infinite number of different sequences which contain the above one as a subsequence. One might think to restrict the domain of possible answers considerably by demanding that a rule be prescribed for generating an arbitrary number of successors of the given subsequence. But such is not the case -- in fact, for any given finite subsequence, there are an infinite number of such rules. Thus, in the example given, the intended rule might be - each term is the integer successor of the last, modulo 6. In that case the next term would be Ø. But the hapless subject might argue with some vehemence that a far more *natural* generation rule is the ordinary integer successor -- which would prescribe 6 as the next term.

Most people feel they have a good idea of what naturalness means. But, how can we impart the same sense to a computer? To resolve this non-mathematical problem -- of choosing the most natural rules -- we let the participants in our game make up the rules for generating their problem sequences and thereby define their criteria of naturalness operationally, by the choices they actually make. Thus, the computer "learns" their meaning of naturalness in an adaptive, evolutionary manner. Our problem then is to design a computer program which, using rules defined by its users, can build effective and natural guessing strategies.

We wish to solve not only "easy" problems, like that in the above example, but also sequences of moderate difficulty such as

-3 1 13 33 61 97

2.2 Designing a Sequence Extrapolation Program

We wish to design a program for predicting the next term of any sequence given to it as an input. Our object is not to make a "perfect" program that would correctly extrapolate any sequence given to it by a mathematician, but only to do so for most sequences devised by high-school students. A conceptually simple way of proceeding is to gradually build up in memory a very large storehouse of sequences, a sort of "sequence bank", and directly look for a copy of the given sequence there so as to find its possible continuation.

The program starts out with an empty sequence bank. In this initial state it must immediately give up on its first problem and ask for the solution. It then stores the entire sequence, including the new term, as the first entry in its sequence bank. The next time it is given a sequence, it can check whether or not this new sequence is a subsequence of one it already contains. If so, it can hazard a possible extrapolation. If not, or if the extrapolation is incorrect, it asks for the solution and stores this new sequence too.

If we think about implementing this scheme, we find that it has some very serious drawbacks. Using it, we must exclude many of the natural sequences of interest. For example, to store just the *constant* sequences (e.g., 1 1 1 1 1) of length 5 for numbers between Ø and 1ØØØ would require nearly 15,000 digits. By comparison, the universal rule for generating the successor for any constant sequence is very easy to write as a program and

this requires no more than a few dozen characters. Furthermore, in addition to the enormous memory requirements for storing sequences, the time required to search through tens of thousands of words of storage becomes prohibitive.

How much more reasonable it would be to store the rule itself, instead of some partial sequence it describes. Not only will this be more efficient -- but we will be able to build more powerful strategies using rules than we could using sequences. Let's, therefore, see what is involved in building an extrapolation program based upon rules.

2.3 Representing Sequences and Generation Rules

Let's consider first how we want to express sequences and generation rules in our programs. Sequences can be represented in LOGO the same way they are ordinarily written, i.e., as LOGO number sentences, for example:

```
"1 2 3 4 5"
"Ø Ø Ø Ø Ø Ø Ø"
"24 48 96"
"14Ø29 871432 -82 4ØØ72"
"4"
```

Rules for prescribing sequences, however, can be of many different forms. One standard form of rule gives for any counting number N, the Nth term of the series. Thus, the sequence

1 2 3 4 5 ...

is described, using this form of "indexing" rule, as "the Nth term is the number N". Another standard form of rule, a kind of successor function, prescribes the *next* term of the sequence given only the previous one. The above sequence can be described by the succession rule "the next term is the sum of the last term and 1". This same rule would apply to -

395 396 397 398
1ØØ2 1ØØ3 1ØØ4
-3 -2 -1 Ø 1

whereas each of these would require a different indexing rule than the one given above.

Clearly, we would like to regard all of these sequences as subsequences subsumed under a single rule. So we will use successor functions for our standard way of prescribing and generating sequences. The natural representation for a successor function in LOGO is a single-input LOGO procedure. The input to the procedure is a term in a sequence and the output is the next term in that sequence. Thus, for the sequence 1 2 3 4 5, and the related ones just shown, the LOGO successor procedure can be written:

```
TO ADD-ONE /N/
1Ø OUTPUT SUM OF /N/ AND 1
END
```

This kind of successor function is not universal -- there are sequences that cannot be described by it. An obvious class of such sequences includes those whose terms are functions of two or more preceding terms; for example, sequences which oscillate as follows.

Ø 1 2 3 4 5 4 3 2 1 Ø 1 2 ...

It is not possible to write a one-input successor procedure for this sequence since a term is not always determined by its immediate predecessor. The successor of 2, for example, can be either 3 or 1, depending on whether that 2 is in the increasing or decreasing part of the sequence.

Nevertheless, the successor function is a good choice for a first form of sequence generation rule. This kind of rule is both easy

to describe and sufficient for describing many different kinds of sequences. It can, in fact, be used for most sequences students are likely to think of. Let's limit ourselves initially then to integer sequences whose terms can be derived from the previous term alone.

2.4 Testing Successor Procedures on Sequences

We can test whether or not a successor procedure describes a sequence by trying it out on the known terms. If ADD-ONE is truly a successor function for the sequence "1 2 3 4 5", then it must correctly output the second term given the first, the third given the second, etc., for all the terms given, as follows.

```
←PRINT ADD-ONE OF 1
2
←PRINT ADD-ONE OF 2
3
←PRINT ADD-ONE OF 3
4
←PRINT ADD-ONE OF 4
5
←
```

Let's write a procedure TESTER to perform this test of a successor's possible validity. TESTER needs two inputs, a sequence and the name of a successor-procedure. It compares the output of the given procedure with the next term, or each pair of terms, in the sequence until either the successor fails or the sequence is exhausted.

```
TO TESTER /SEQUENCE/ AND /PROCEDURE-NAME/
1Ø TEST EMPTYP OF BUTFIRST OF    (Have all the pairs of terms
    /SEQUENCE/                    been tested?)
2Ø IF TRUE OUTPUT "TRUE"         (If so, the test is successful)
3Ø TEST IS SECOND OF /SEQUENCE/  (Is the successor of the first
    EXECUTE OF /PROCEDURE-NAME/   term identical to the second
    AND FIRST OF /SEQUENCE/       term?)
4Ø IF FALSE OUTPUT "FALSE"       (If not, the successor does not
                                  describe the sequence)
5Ø OUTPUT TESTER OF
    (BUTFIRST OF /SEQUENCE/)     (Otherwise go on to the rest
    AND /PROCEDURE-NAME/          of the sequence)
END
```

TESTER uses two subprocedures, SECOND and EXECUTE. SECOND simply outputs the second term of a given sequence.

```
TO SECOND /SEQUENCE/
1Ø OUTPUT FIRST OF BUTFIRST OF /SEQUENCE/
END
```

EXECUTE is the procedure that actually applied the successor procedure to the terms in the sequence. It uses the DO command to perform the procedure named by /PROCEDURE-NAME/:

```
TO EXECUTE /PROCEDURE/ AND /INPUT/
1Ø DO SENTENCE OF "OUTPUT" AND SENTENCE
    OF /PROCEDURE/ AND /INPUT/
END
```

e.g.,

```
←EXECUTE "ADD-ONE" "5"
6
←
```

TESTER is used as follows.

```
←PRINT TESTER OF "1 2 3 4 5" AND "ADD-ONE"
TRUE
←PRINT TESTER OF "1 3 5 7 9" AND "ADD-ONE"
FALSE
←
```

Let's introduce another successor procedure now.

```
TO ADD-TWO /N/
1Ø OUTPUT SUM OF /N/ AND 2
END

←PRINT TESTER OF "1 3 5 7 9" AND "ADD-TWO"
TRUE
←
```

Given a list of successor procedures stored in a "procedure bank", TESTER can be used to determine which of these procedures describes a given sequence. If one or more of them are successful, we can consider using these for predicting the next term of the sequence.

Assuming we have such a /PROCEDURE-BANK/ (i.e., a sentence of successor-procedure names), the next thing we want is a procedure to apply TESTER with a given sequence on each of the procedures listed in /PROCEDURE-BANK/. This new procedure SCAN-P-LIST will scan the list of procedures, apply each one in succession, and output the name of the first procedure that works. If none of the successors describe the sequence, SCAN-P-LIST will output /EMPTY/.

```
TO SCAN-P-LIST /SEQUENCE/ AND /PROCEDURES/
1Ø TEST EMPTYP /PROCEDURES/      (Have we exhausted the list of
                                  possible successor-procedures?)
2Ø IF TRUE OUTPUT /EMPTY/        (If so, we have no effective
                                  procedure)
3Ø TEST TESTER OF /SEQUENCE/     (Does the first procedure describe
    AND (FIRST OF /PROCEDURES/)   the sequence?)
4Ø IF TRUE OUTPUT FIRST OF       (If so, make it our candidate)
    /PROCEDURES/
5Ø OUTPUT SCAN-P-LIST OF         (Otherwise, try the rest of the
    /SEQUENCE/ AND (BUTFIRST OF   list)
    /PROCEDURES/)
END
```

Let's build a /PROCEDURE-BANK/ and try out SCAN-P-LIST:

```
←MAKE "PROCEDURE-BANK" "ADD-ONE ADD-TWO"
←PRINT SCAN-P-LIST OF "1 2 3 4 5" AND /PROCEDURE-BANK/
ADD-ONE
←PRINT SCAN-P-LIST OF "1 3 5 7 9" AND /PROCEDURE-BANK/
ADD-TWO
←PRINT SCAN-P-LIST OF "2 4 8 16 32" AND /PROCEDURE-BANK/
                              (SCAN-P-LIST outputs /EMPTY/)
←PRINT SCAN-P-LIST OF "1" AND /PROCEDURE-BANK/
ADD-ONE
←
```

The last example points to an obvious limitation in SCAN-P-LIST in its current form. The successor procedure ADD-TWO describes the given sequence as well as does ADD-ONE and, indeed, the extrapolated sequence might be "1 3 5 7 ..." instead of "1 2 3 4 ...". But SCAN-P-LIST has stopped short of any further testing of ADD-ONE that might rule it in or out as the actual solution. The requirement that a successor procedure satisfy TESTER for all terms given in the input sequence is not sufficient; the procedure should also extrapolate a correct next term, as judged by the user. Unless one of the procedures in /PROCEDURE-BANK/ can satisfy both these conditions, SCAN-P-LIST should keep searching until the bank is exhausted.

Once a successor procedure is found that satisfies TESTER on all the given terms, we can easily compute the next term given by the procedure, as follows.

```
MAKE "NEXT-TERM" EXECUTE OF (FIRST OF /PROCEDURES/)
 AND (LAST OF /SEQUENCE/)
```

We must then ask the user whether this is, in fact, the *correct* next term. If it is the number he intends, we regard the

sequence as solved.* If not, we continue scanning the /PROCEDURE-BANK/ for other possible solutions. Assuming we write a sub-procedure, APPROVAL, that elicits the user's acceptance of the tentative next term and outputs TRUE or FALSE accordingly, SCAN-P-LIST now looks like this:

```
TO SCAN-P-LIST /SEQUENCE/ AND /PROCEDURES/
1Ø TEST EMPTYP /PROCEDURES/
2Ø IF TRUE OUTPUT /EMPTY/
3Ø TEST TESTER OF /SEQUENCE/ AND (FIRST OF /PROCEDURES/)
4Ø IF FALSE OUTPUT SCAN-P-LIST OF /SEQUENCE/ AND
    (BUTFIRST OF /PROCEDURES/)
5Ø MAKE "NEXT-TERM" EXECUTE OF (FIRST OF /PROCEDURES/)
    AND (LAST OF /SEQUENCE/)
6Ø TEST APPROVAL OF /NEXT-TERM/
7Ø IF TRUE OUTPUT (FIRST OF /PROCEDURES/)
8Ø OUTPUT SCAN-P-LIST OF /SEQUENCE/ AND
    (BUTFIRST OF /PROCEDURES/)
END
```

The subprocedure APPROVAL need only type out the tentative next term and test the user's response:

```
TO APPROVAL /NEXT-TERM/
1Ø TYPE SENTENCE OF "IS THE NEXT TERM" AND SENTENCE OF
    /NEXT-TERM/ AND "?..."        (Ask for user's judgment of
2Ø MAKE "ANS" REQUEST              tentative next term)
3Ø TEST IS /ANS/ "YES"
4Ø IF FALSE OUTPUT "FALSE"        (Rejection)
5Ø OUTPUT "TRUE"                  (Acceptance)
END
```

*Of course, he may have another sequence in mind which is identical to this one up to this number of terms. Assume for example that a user has in mind the sequence "1 2 4 16 ..." (each number is 2 raised to the previous number). To exclude "1 2 4 8 ..." (each number is 2 times the previous one) from being declared the solution, he merely has to input the starting sequence "1 2 4", since "1 2" is not sufficient. It is incumbent on the user to give enough terms to "uniquely specify" the solution.

We can now try our modified SCAN-P-LIST on /PROCEDURE-BANK/:

```
←PRINT SCAN-P-LIST OF "1 2 3 4" AND /PROCEDURE-BANK/
IS THE NEXT TERM 5?...YES          (The extrapolation is acceptable)
ADD-ONE                      (So SCAN-P-LIST announces the solution)
←PRINT SCAN-P-LIST OF "1" AND /PROCEDURE-BANK/
IS THE NEXT TERM 2?...NO           (This extrapolation is rejected)
IS THE NEXT TERM 3?...YES          (This one is accepted)
ADD-TWO                            (SCAN-P-LIST gives the solution)
←PRINT SCAN-P-LIST OF "2 4 8 16"
 AND /PROCEDURE-BANK/
                                   (No solution was found)
←
```

2.5 Enlarging the Procedure Bank

The power of this extrapolation program depends entirely on the contents of its /PROCEDURE-BANK/ and thus far we have no means of extending it. A good way to do this is to ask the user to supply us with the successor procedures for any sequence SCAN-P-LIST can't solve. If the procedure he types in does indeed describe his sequence, we can add its name to /PROCEDURE-BANK/. The next time we use the extrapolation program the new procedure will be considered along with the others.

The procedure OCCAM sets up SCAN-P-LIST and requests a successor procedure from the user if SCAN-P-LIST fails.

```
TO OCCAM
1Ø TYPE "WHAT IS YOUR SEQUENCE?..."
2Ø MAKE "SEQUENCE" REQUEST
3Ø TEST EMPTYP OF SCAN-P-LIST OF /SEQUENCE/
    AND /PROCEDURE-BANK/          (Note that OCCAM does not announce
4Ø IF FALSE STOP                  the name of the winning procedure)
5Ø PRINT "I CAN'T DO THAT ONE.  WRITE A LOGO PROCEDURE THAT
    WILL FIND THE SUCCESSOR OF ANY TERM IN THE SEQUENCE AND
    THEN TYPE 'CONTINUE'"
END
```

CONTINUE is a LOGO procedure that asks for the name of the user's procedure, and then tests this procedure on the sequence (using TESTER). If the procedure correctly describes the sequence, OCCAM adds the procedure to /PROCEDURE-BANK/; otherwise, it executes the procedure and prints out the sequence it actually generates.

```
TO CONTINUE
1Ø TYPE "WHAT WAS THE NAME OF YOUR PROCEDURE?..."
2Ø MAKE "PROCEDURE" REQUEST
3Ø TEST TESTER OF /SEQUENCE/ AND /PROCEDURE/
4Ø IF TRUE MAKE "PROCEDURE-BANK" (SENTENCE OF
    /PROCEDURE-BANK/ AND /PROCEDURE/)
5Ø IF FALSE PRINT SENTENCE OF SENTENCE OF
    "ERROR ON YOUR PROCEDURE.  I RAN IT STARTING WITH"
    AND FIRST OF /SEQUENCE/ AND "AND GOT:"
6Ø IF FALSE RUN /PROCEDURE/ AND (FIRST OF /SEQUENCE/)
END
```

The subprocedure RUN prints the sequence described by the faulty successor procedure indefinitely:

```
TO RUN /PROCEDURE/ AND /FIRST-TERM/
1Ø PRINT /FIRST-TERM/
2Ø MAKE "NEXT-TERM" EXECUTE OF /PROCEDURE/
    AND /FIRST-TERM/
3Ø RUN /PROCEDURE/ AND /NEXT-TERM/
END
```

We now have a sequence extrapolation procedure that can automatically be extended with use. To begin with, it knows how to handle two kinds of sequences:

```
←OCCAM
WHAT IS YOUR SEQUENCE?...7 8 9 1Ø
IS THE NEXT TERM 11?...YES
←OCCAM
WHAT IS YOUR SEQUENCE?...22 24 26 28
IS THE NEXT TERM 3Ø?...YES
←
```

And it will quickly need to incorporate new ones:

```
←OCCAM
WHAT IS YOUR SEQUENCE?...2 4 8 16
I CAN'T DO THAT ONE.  WRITE A LOGO PROCEDURE THAT
WILL FIND THE SUCCESSOR OF ANY TERM IN THE SEQUENCE
AND THEN TYPE 'CONTINUE'
←TO TIMES-TWO /N/
>1Ø OUTPUT PRODUCT OF /N/ AND 2
>END
TIMES-TWO DEFINED
←CONTINUE
WHAT WAS THE NAME OF YOUR PROCEDURE?...TIMES-TWO
←OCCAM
WHAT IS YOUR SEQUENCE?...5 1Ø 2Ø
IS THE NEXT TERM 4Ø?...YES
←
```

Now that things are beginning to move, let's make some changes to shorten OCCAM's printouts and reduce the user's typing somewhat.

```
←EDIT OCCAM
>1Ø TYPE "SEQUENCE?..."
>5Ø PRINT "CAN'T DO THAT ONE.  TELL ME HOW AND THEN
     TYPE 'CONTINUE'"
>END
OCCAM DEFINED
←EDIT CONTINUE
>1Ø TYPE "PROCEDURE NAME?..."
>END
CONTINUE DEFINED
←ABBREVIATE "CONTINUE" AS "CON"
←
```

And now we'll try a new kind of sequence.

```
←OCCAM
SEQUENCE?...1 4 9 16
CAN'T DO THAT ONE.  TELL ME HOW AND THEN TYPE 'CONTINUE'
←TO SQUARES /N/
>1Ø MAKE "X" SUM OF (SQUARE-ROOT OF /N/) AND 1   (Assuming we've
>2Ø OUTPUT PRODUCT OF /X/ AND /X/                written a SQUARE-
>END                                             ROOT procedure)
←CON
PROCEDURE NAME?...SQUARES
←
```

```
←OCCAM
SEQUENCE?...25 36 49
IS THE NEXT TERM 64?...YES
←
```

Let's check the other branch of CONTINUE by making a deliberate mistake.

```
←OCCAM
SEQUENCE?...1 8 14 21
CAN'T DO THAT ONE.  TELL ME HOW AND THEN TYPE 'CONTINUE'
←TO ADD-SEVEN /N/
>1Ø OUTPUT SUM OF /N/ AND 7
>END
←CON
PROCEDURE NAME?...ADD-SEVEN
ERROR IN YOUR PROCEDURE.  I RAN IT STARTING WITH 1 AND GOT:
1
8
15
22
29
BREAK                    (The user terminates the computation here)
I WAS AT LINE 1Ø IN EXECUTE
←
```

Let's also see how OCCAM handles ambiguous sequences.

```
←OCCAM
SEQUENCE?...9 16
IS THE NEXT TERM 25?...NO
IS THE NEXT TERM 23?...YES
←
```

Another interesting kind of sequence, that involves a non-numerical successor function, is added next.

```
←OCCAM
SEQUENCE?...1 11 111 1111
CAN'T DO THAT ONE.  TELL ME HOW AND THEN TYPE 'CONTINUE'
←TO GROW-ONE /N/
>1Ø OUTPUT WORD OF /N/ AND "1"
>END
←
```

Unfortunately, OCCAM can't extend its knowledge of GROW-ONE to get this sequence:

```
←OCCAM
SEQUENCE?...2 22 222
CAN'T DO  [etc.]
←TO GROW-TWO /N/
>1Ø OUTPUT WORD OF /N/ AND "2"
>END
GROW-TWO DEFINED
←CON
PROCEDURE NAME?...GROW-TWO
←OCCAM
SEQUENCE?...1 12
IS THE NEXT TERM 122?...YES
←
```

We can include sequences whose successive terms get smaller instead of larger.

```
←OCCAM
SEQUENCE?...6 5 4
CAN'T DO ...
←TO DIM-ONE /N/
>1Ø OUTPUT DIFFERENCE OF /N/ AND 1
>END
DIM-ONE DEFINED
←CON
PROCEDURE NAME?...DIM-ONE
←
```

Also, we can include sequences that are not monotone:

```
←OCCAM
SEQUENCE?...Ø 1 2 3 Ø 1 2
CAN'T DO ...
←TO MOD-FOUR /N/
>1Ø OUTPUT REMAINDER OF (SUM OF
     /N/ AND 1) AND 4
>END
MOD-FOUR DEFINED
←
```

With some effort we can even add the factorial sequence (where the factorial function is defined F(N) = N·F(N-1), N>1; F(1)=1)

```
←OCCAM
SEQUENCE?...1 2 6 24
CAN'T DO ...
←TO NEXT-FACTORIAL /N/
>1Ø OUTPUT FACT OF /N/ AND 1     (NEXT-FACTORIAL uses the following
>END                              recursive subprocedure, FACT)
NEXT-FACTORIAL DEFINED
←TO FACT /N/ AND /COUNTER/
>1Ø TEST GREATERP /COUNTER/ AND /N/ (Is /COUNTER/ bigger than /N/?)
>2Ø IF TRUE OUTPUT /COUNTER/        (If so, the answer is /COUNTER/)
>3Ø OUTPUT PRODUCT OF /COUNTER/ AND (If not, the answer is
     FACT OF (QUOTIENT OF /N/ AND    /COUNTER/ multiplied by the
     /COUNTER/) AND (SUM OF /COUNTER/ result of the indicated
     AND 1)                          recursion)
>END
FACT DEFINED
←CON
PROCEDURE NAME?...NEXT-FACTORIAL
←OCCAM
PROCEDURE?...6 24
IS THE NEXT TERM 12Ø?...YES
←
```

Our extrapolation program is beginning to acquire some power. But, it has some unnecessary operational flaws. For instance, it can make annoyingly repetitive extrapolations. As an example, look at what happens now if our input sequence is simply "1".

```
IS THE NEXT TERM 2?...NO
IS THE NEXT TERM 3?...NO
IS THE NEXT TERM 2?...NO
IS THE NEXT TERM 4?...NO
IS THE NEXT TERM 8?...NO
IS THE NEXT TERM 11?...NO
IS THE NEXT TERM 12?...NO
IS THE NEXT TERM Ø?...NO
IS THE NEXT TERM 2?...NO
```

We see here that 2 was offered as a next term three times. Let's modify OCCAM so that it will never try any /NEXT-TERM/ more than

once. All we need do is keep a running list of all the /NEXT-TERM/s that failed, and check to see whether or not a /NEXT-TERM/ is already on the list before we submit it to the user.

This list, call it /BAD-NEXT-TERMS/, shculd be made empty each time OCCAM is used. One new line does this.

```
←EDIT OCCAM
>5 MAKE "BAD-NEXT-TERMS" /EMPTY/
>END
OCCAM DEFINED
←
```

Related changes must be made to APPROVAL. First of all, APPROVAL should check to see if its input /NEXT-TERM/ is on this list of rejected extrapolations. It uses a subprocedure CONTAINSP, defined below, to do this test.

```
←EDIT APPROVAL
>5 TEST CONTAINSP /BAD-NEXT-TERMS/ /NEXT-TERM/
```

If /NEXT-TERM/ is on the list, we know without further query that it is not the correct next term. Thus:

```
>6 IF TRUE OUTPUT "FALSE"
```

Also, APPROVAL should add any unsuccessful extrapolations to the list as soon as they have been rejected:

```
>35 IF FALSE MAKE "BAD-NEXT-TERMS" (SENTENCE
    OF /BAD-NEXT-TERMS/ AND /NEXT-TERM/)
>END
APPROVAL DEFINED
```

The procedure CONTAINSP is a predicate with two inputs, a list and an element.

```
TO CONTAINSP /LIST/ AND /ELEMENT/
1Ø TEST EMPTYP /LIST/              (Have we exhausted the list?)
2Ø IF TRUE OUTPUT "FALSE"          (If so, the list did not contain
                                    the element)
3Ø TEST IS (FIRST OF /LIST/)       (Is the first thing on /LIST/
    /ELEMENT/                       the element?)
4Ø IF TRUE OUTPUT "TRUE"           (Yes, /LIST/ contains /ELEMENT/)
5Ø OUTPUT CONTAINSP OF (BUTFIRST   (Otherwise, try the rest of
    OF /LIST/) AND /ELEMENT/        the list)
END
```

OCCAM's responses to the sequence "1" are now non-repetitive:

```
IS THE NEXT TERM 2?...NO
IS THE NEXT TERM 3?...NO
IS THE NEXT TERM 4?...NO
IS THE NEXT TERM 8?...NO
IS THE NEXT TERM 11?...NO
IS THE NEXT TERM 12?...NO
IS THE NEXT TERM Ø?...NO
CAN'T DO ...
```

2.6 Automatic Generalization of Successor Procedures

OCCAM has another more basic limitation. If we look at the current procedure-bank --

```
←PRINT /PROCEDURE-BANK/
ADD-ONE ADD-TWO TIMES-TWO SQUARES ADD-SEVEN GROW-ONE
GROW-TWO DIM-ONE MOD-FOUR NEXT-FACTORIAL
←
```

we find many successor procedures that are virtually identical in form. In particular, the LOGO definitions for the procedures ADD-ONE, ADD-TWO, and ADD-SEVEN differ only by a constant. (The same is true for GROW-ONE and GROW-TWO.) These are essentially the same procedures. Moreover, many more procedures of the same kind will surely be added as OCCAM is used. Clearly it is inefficient to have so many instances of virtually the same generating procedures. More importantly, though, no matter how many instances of the same general form OCCAM has seen, it will

nevertheless be unable to extrapolate any new such instance. In effect, it fails to generalize its experience. If OCCAM somehow were able to construct, from a single such instance, a more general procedure which included *all* similar instances, it would gain enormously in power. In particular, it would then be able to extrapolate sequences it has never seen before. There is a simple way of accomplishing this for many procedures of interest.

Let's redefine ADD-ONE as follows.

```
TO ADD-ONE /N/ AND /DUMMY/          (/DUMMY/ is a new input)
1Ø OUTPUT SUM OF /N/ AND /DUMMY/ (/DUMMY/ replaces the old "1"
END                                  here)
```

The original ADD-ONE describes a particular arithmetic sequence. Replacing the constant "1" with a dummy variable has the effect of generalizing the original one-input successor function to a two-input successor which describes the entire family of arithmetic sequences. If /DUMMY/ is 1, we get the original ADD-ONE again; with /DUMMY/ set to 2, we have ADD-TWO, and so on for all such arithmetic sequences (i.e., those whose successive terms differ by some constant). Similarly, a generalized TIMES-TWO

```
TO TIMES-TWO /N/ AND /DUMMY/
1Ø OUTPUT PRODUCT OF /N/ AND /DUMMY/
END
```

will describe all *geometric sequences*, sequences whose terms are some constant multiple of their predecessor. Let's show next how OCCAM can construct these generalized procedures. Then we will see how it can use them to improve its extrapolation capabilities. In the examples we looked at, ADD-ONE and TIMES-TWO, the generalization was accomplished by replacing the constant with a variable, /DUMMY/, and appending /DUMMY/ to the title. Some procedures contain two or more constants that could be replaced by variables but, instead of replacing all of these,

we will content ourselves with simply replacing the *first* constant that occurs within the procedure.

To change a one-input successor procedure into such a "generalized" two-input form, we will need to be able to write procedures for modifying other procedures. We can do this using the LOGO command DO along with two LOGO operations, LINES and TEXT, which enable us to extract from any procedure the list of line numbers and the associated instructions in specified lines. The LOGO operation LINES takes as its input a procedure name and outputs a sentence made up of the line numbers contained in the procedure, for example:

```
←PRINT LINES OF "APPROVAL"
5 6 1Ø 2Ø 3Ø 35 4Ø 5Ø
←
```

The built-in operation TEXT gives us access to the instruction lines themselves. For example:

```
←PRINT TEXT OF "APPROVAL" AND 5Ø
5Ø OUTPUT "TRUE"
←
```

With access to the contents of a procedure, we can write a procedure SCAN-LINES which goes through a procedure line-by-line looking for a constant to replace.

```
TO SCAN-LINES /PROCEDURE/ AND /LINES/  (LINES is the sentence of
                                         line numbers)
1Ø TEST EMPTYP /LINES/          (Have we gone through all the lines?)
2Ø IF TRUE OUTPUT "FALSE"       (If so, there were no constants to
                                  replace)
3Ø MAKE "LINE-TEXT" TEXT OF       (Get the text of the first line)
    /PROCEDURE/ AND (FIRST
    OF /LINES/)
4Ø TEST CONTAINS-NUMBER OF        (Are there any numbers in the
    BUTFIRST OF /LINE-TEXT/        line besides the line number?)
5Ø IF FALSE OUTPUT SCAN-LINES   (If not, check the next line)
    OF /PROCEDURE/ AND BUTFIRST
    OF /LINES/
```

```
6Ø CHANGE /PROCEDURE/ AND          (If so, edit the procedure)
    /LINE-TEXT/                    (and indicate that we have
7Ø OUTPUT "TRUE"                    generalized the procedure)
END
```

The subprocedure CONTAINS-NUMBER simply applies NUMBERP to every word (excluding the line number) in the text of an instruction line:

```
TO CONTAINS-NUMBER /LINE/
1Ø TEST EMPTYP /LINE/              (Have we exhausted the line?)
2Ø IF TRUE OUTPUT "FALSE"          (If so, it contained no constant)
3Ø TEST NUMBERP OF FIRST OF /LINE/  (Is first word a number?)
4Ø IF TRUE OUTPUT "TRUE"           (The answer is yes)
5Ø OUTPUT CONTAINS-NUMBER OF       (If not, repeat the process with
    (BUTFIRST OF /LINE/)            the rest of the line)
END
```

SCAN-LINES uses CHANGE to edit a procedure known to contain a constant. (The LOGO command DO takes its input as a LOGO instruction and executes it as such.)

```
TO CHANGE /PROCEDURE/ AND /LINE-TEXT/
1Ø DO SENTENCE OF "EDIT" AND /PROCEDURE/   (Put LOGO into edit mode)
2Ø DO SENTENCE OF (TEXT OF /PROCEDURE/     (Append /DUMMY/ onto
    AND Ø) (AND "/DUMMY/")                  line Ø, the title line)
3Ø DO SENTENCE OF FIRST OF
    /LINE-TEXT/ AND REWRITE OF             (Rewrite the line contain-
    BUTFIRST OF /LINE-TEXT/                 ing a number)
4Ø DO "END"                                (Leave edit mode)
END
```

CHANGE's subprocedure, REWRITE, is very much like CONTAINS-NUMBER; it scans a line looking for a constant. However, REWRITE actually reconstructs the line, changing the first number in it to the word "/DUMMY/".

```
TO REWRITE /LINE/
1Ø TEST NUMBERP OF FIRST OF      (Is the first word a number?)
    /LINE/
2Ø IF TRUE OUTPUT SENTENCE OF    (If so, replace it with "/DUMMY/"
    "/DUMMY/" AND BUTFIRST OF     and append the rest of the line)
    /LINE/
3Ø OUTPUT SENTENCE OF (FIRST OF (Otherwise, put it back in the
    /LINE/) AND (REWRITE OF       line and repeat the process
    BUTFIRST OF /LINE/)           with the rest of the line)
END
```

The outer procedure GENERALIZE simply invokes SCAN-LINES, giving it a procedure name and the associated list of the procedure's line numbers:

```
TO GENERALIZE /PROCEDURE/
1Ø OUTPUT SCAN-LINES OF /PROCEDURE/ AND
    (LINES OF /PROCEDURE/)
END
```

Let's try out these new generalization capabilities:

```
←PRINT GENERALIZE OF "ADD-ONE"  (ADD-ONE has now been generalized)
TRUE
←LIST ADD-ONE
TO ADD-ONE /N/ /DUMMY/
1Ø OUTPUT SUM OF /N/ AND /DUMMY/
END
←
```

Not all successor procedures can be generalized in this way. GENERALIZE acknowledges its inability to generalize a procedure with the output "FALSE". For example:

```
←TO SQUARE /N/
1Ø OUTPUT PRODUCT OF /N/ AND /N/
END
SQUARE DEFINED
←PRINT GENERALIZE OF "SQUARE"
FALSE
←
```

Now that we can construct some generalized procedures, let's see how to incorporate these in our extrapolation program.

SCAN-P-LIST and TESTER, as they stand, will not work for two-input successor procedures. We still need these procedures in their present form to handle the ungeneralizable successors in /PROCEDURE-BANK/, but we will also require precisely parallel procedures to do the same things for generalized successor procedures that we will accumulate in /GENERAL-BANK/.

First, let's write a new TESTER. To determine whether or not a general successor correctly describes a given sequence, we will have to test it on every term across a range of values for /DUMMY/. A reasonable way to do this is to divide the problem into two parts -- to write one procedure, GENERAL-TESTER, to run through the range of /DUMMY/ values, and another, GEN-TEST-2 to check each pair of known terms in the problem-sequence with some fixed value of /DUMMY/. GEN-TEST-2 is almost identical to TESTER.

```
TO GEN-TEST-2 /SEQUENCE/ AND /PROCEDURE/ AND /DUMMY/
1Ø TEST EMPTYP BUTFIRST OF /SEQUENCE/  (Have we checked all the
                                         pairs of terms?)
2Ø IF TRUE OUTPUT "TRUE"            (If so, the procedure - with this
                                     /DUMMY/ - is accepted)
3Ø TEST IS (SECOND OF /SEQUENCE/) (Does the procedure correctly
    (EXECUTE-2 OF /PROCEDURE/       give the second term of the
    AND FIRST OF /SEQUENCE/         sequence?)
    AND /DUMMY/
4Ø IF FALSE OUTPUT "FALSE"          (If not, it is rejected)
5Ø OUTPUT GEN-TEST-2 OF BUTFIRST    (If so, repeat the test on
    OF /SEQUENCE/ AND /PROCEDURE/    the rest of the sequence)
    AND /DUMMY/
END
```

The subprocedure EXECUTE-2 is a simple extension of EXECUTE designed to execute two-input procedures:

```
TO EXECUTE-2 /PROCEDURE/ AND /INPUT-1/ AND /INPUT-2/
1Ø DO SENTENCE OF "OUTPUT" AND SENTENCE OF /PROCEDURE/
    AND SENTENCE OF /INPUT-1/ AND /INPUT-2/
END
```

Now we will have to choose a range of values to use with GEN-TEST-2. Should we try two values? ten? one hundred? a thousand? On the one hand, we want to use a "big" range; the larger the range the more effective the successor procedures will be. But, even if it checks all values from Ø to 1ØØ£, a generalized ADD-ONE, for example, would miss the sequence -

1 1ØØ2 2ØØ3 3ØØ4

But a person confronted with such a sequence of large numbers would not carry out this kind of generalization technique for 1ØØØ successive values of a parameter. Rather, he would try to transform the problem into a more tractable form where rules using small parameter values might be effective. Perhaps the most natural such transformation is to replace the original sequence by one consisting of the differences between successive terms of the original -- obviously a good choice for the above sequence. Use of such transformations can vastly improve our extrapolation capabilities. We will develop some procedures of this kind after seeing the kind of benefits we obtain with GENERALIZE.

Since we are not depending upon GENERALIZE to solve problems with large parameters, and since the larger the range chosen the longer it takes to do the checking, we shall choose a small range of values. The range zero through ten is a natural one for people doing parametric calculations so let's make the same choice for our program. GENERAL-TESTER then will test a given generalized procedure on a sequence using values of /DUMMY/ between Ø and 1Ø successively. It will output the first /DUMMY/ that is successful across all known terms in the sequence. If no /DUMMY/ is successful, it will output /EMPTY/.

```
TO GENERAL-TESTER /SEQUENCE/ AND /PROCEDURE/
1Ø MAKE "DUMMY" Ø
2Ø TEST GEN-TEST-2 OF /SEQUENCE/ AND /PROCEDURE/ AND /DUMMY/
3Ø IF TRUE OUTPUT /DUMMY/
4Ø TEST IS /DUMMY/ 1Ø
5Ø IF TRUE OUTPUT /EMPTY/
6Ø MAKE "DUMMY" (SUM OF /DUMMY/ AND 1)
7Ø GO TO LINE 2Ø
END
```

Let's try GENERAL-TESTER with our generalized ADD-ONE:

```
←PRINT GENERAL-TESTER OF "1 5 9 13 17" AND "ADD-ONE"
4                                  (This is the "winning" /DUMMY/)
←PRINT GENERAL-TESTER OF "2 4 8 16" AND "ADD-ONE"
                                   (There is no winning /DUMMY/)
←
```

We will also need a procedure similar to SCAN-P-LIST which uses GENERAL-TESTER to scan a list of generalized successor procedures. The only differences between this procedure, call it SCAN-G-LIST, and the old SCAN-P-LIST are due to the extra input, /DUMMY/; SCAN-G-LIST will have to save the output of GENERAL-TESTER (either a number or /EMPTY/, rather than "TRUE" or "FALSE") and use it in EXECUTE-2 to calculate the next term.

```
TO SCAN-G-LIST /SEQUENCE/ AND /PROCEDURES/
1Ø TEST EMPTYP /PROCEDURES/      (Have we exhausted the list?)
2Ø IF TRUE OUTPUT /EMPTY/        (Yes, no winners in the list)
3Ø MAKE "WINNING-DUMMY" GENERAL-TESTER  (Test the first procedure
    OF /SEQUENCE/ AND (FIRST OF          - save the result)
    /PROCEDURES/)
4Ø TEST EMPTYP /WINNING-DUMMY/     (Did the procedure fail?)
5Ø IF TRUE OUTPUT SCAN-G-LIST OF   (If so, try the rest of the
    /SEQUENCE/ AND (BUTFIRST OF     list)
    /PROCEDURES/)
6Ø MAKE "NEXT-TERM" EXECUTE-2 OF   (If the procedure succeeded,
    FIRST OF /PROCEDURES/ AND       use it to calculate a next
    LAST OF /SEQUENCE/ AND          term)
    /WINNING-DUMMY/
7Ø TEST APPROVAL OF /NEXT-TERM/    (Is this extrapolation accepted?)
8Ø IF TRUE OUTPUT FIRST OF /PROCEDURES/  (Yes)
9Ø OUTPUT SCAN-G-LIST OF /SEQUENCE/  (No, so repeat the process
    AND (BUTFIRST OF /PROCEDURES/)    with the rest of the proce-
END                                   dures)
```

We now have procedures that can generalize successor functions and procedures that can test lists of generalized successors against a given sequence. Linking these procedures to our existing sequence extrapolation is straightforwardly done.

First of all, we need to modify CONTINUE so as to add new procedures to /GENERAL-BANK/. When a new successor satisfies TESTER, CONTINUE should give it to GENERALIZE. If GENERALIZE outputs "TRUE", the generalized procedure can be added to /GENERAL-BANK/, otherwise the procedure is not generalizable and must be appended to our other list, /PROCEDURE-BANK/. With these changes CONTINUE looks as follows.

```
TO CONTINUE
1Ø TYPE "PROCEDURE NAME?..."
2Ø MAKE "PROCEDURE" REQUEST
3Ø TEST TESTER OF /SEQUENCE/ AND /PROCEDURE/
4Ø IF FALSE PRINT SENTENCE OF "ERROR IN YOUR PROCEDURE. I
    RAN IT STARTING WITH" AND SENTENCE OF (FIRST OF
    /SEQUENCE/) AND "AND GOT:"
5Ø IF FALSE RUN /PROCEDURE/ AND (FIRST OF /SEQUENCE/)
6Ø TEST GENERALIZE OF /PROCEDURE/  (Does the procedure generalize?)
7Ø IF TRUE MAKE "GENERAL-BANK"     (If so, add the generalized
    SENTENCE OF /GENERAL-BANK/      procedure to /GENERAL-BANK/)
    AND /PROCEDURE/
8Ø IF FALSE MAKE "PROCEDURE-BANK"  (Otherwise, put it with the
    SENTENCE OF /PROCEDURE-BANK/    other one-input successor)
    AND /PROCEDURE/
END
```

All that remains to do is to give OCCAM the option of using generalized procedures (if there are any) for its extrapolations. So we will have to do SCAN-G-LIST after it does SCAN-P-LIST. This is done by adding a single line to OCCAM.

```
35 IF TRUE TEST EMPTYP OF
    SCAN-G-LIST OF /SEQUENCE/   (If SCAN-P-LIST fails,
    AND /GENERAL-BANK/           test SCAN-G-LIST)
```

2.7 Extrapolation Using Generalized Successor Procedures

We're now able to try the new and more powerful extrapolation program we have been building. Since many of the successor functions in /PROCEDURE-BANK/ can be expressed more efficiently with a generalized successor, let's start once again with a completely blank slate.

```
←MAKE "PROCEDURE-BANK" /EMPTY/
←OCCAM
SEQUENCE?...5 5 5 5
CAN'T DO THAT ONE.  TELL ME NOW AND THEN TYPE 'CONTINUE'
←TO FIVERS /N/
>1Ø OUTPUT 5
>END
FIVERS DEFINED
←CON
PROCEDURE NAME?...FIVERS
←OCCAM
SEQUENCE?...8 8 8 8
IS THE NEXT TERM 8?...YES
←
```

A user who didn't know about our generalizer might be surprised to see OCCAM solving sequences for which it wasn't specifically given a rule. Instead of keeping him completely in the dark about OCCAM's methods though, we might like to let him know what particular procedure(s) it employed to extrapolate his sequence. So let's modify SCAN-P-LIST to declare a "winning" ungeneralized procedure, as follows.

```
65 IF TRUE PRINT SENTENCE OF "THE WINNING PROCEDURE WAS"
     AND (FIRST OF /PROCEDURES/)
```

In modifying SCAN-G-LIST we will want to announce the /WINNING-DUMMY/ as well as the procedure name:

```
75 IF TRUE PRINT SENTENCE OF "THE WINNING PROCEDURE WAS"
     AND SENTENCE OF (FIRST OF /PROCEDURES/) AND SENTENCE
     OF "FOR /DUMMY/ EQUALS" AND /WINNING-DUMMY/
```

Now let's try the last example again:

```
←OCCAM
SEQUENCE?...8 8 8 8
IS THE NEXT TERM 8?...YES
THE WINNING PROCEDURE WAS FIVERS FOR /DUMMY/ EQUALS 8
←LIST FIVERS                          (The original FIVERS procedure
TO FIVERS /N/ /DUMMY/                  was generalized as shown)
1Ø OUTPUT /DUMMY/
END
←
```

The generalized procedure FIVERS can solve all sequences of constant terms (between Ø and 1Ø). We can effect comparable generalizations for arithmetic and geometric sequences as easily.

```
←OCCAM
SEQUENCE?...99 1Ø1 1Ø3 1Ø5
CAN'T DO THAT ONE.  TELL ME HOW AND THEN TYPE 'CONTINUE'
←TO ADD-TWO /N/
←1Ø OUTPUT SUM OF /N/ AND 2
←END
ADD-TWO DEFINED
←CON
PROCEDURE NAME?...ADD-TWO
←OCCAM
SEQUENCE?...99 1Ø5 111
IS THE NEXT TERM 117?...YES
THE WINNING PROCEDURE WAS ADD-TWO FOR /DUMMY/ EQUALS 6
←OCCAM
SEQUENCE?...8 16 32
CAN'T DO THAT ONE.  TELL ME HOW AND THEN TYPE 'CONTINUE'
  : : :                    (TIMES-TWO is entered here)
←CON
SEQUENCE NAME?...TIMES-TWO
←OCCAM
SEQUENCE?...1 9 81 729
IS THE NEXT TERM 6561?...YES
THE WINNING PROCEDURE WAS TIMES-TWO FOR /DUMMY/ EQUALS 9
←
```

We can also add our old procedures DIM-ONE and GROW-ONE, and these will also be generalized.

Then, for example:

```
←OCCAM
SEQUENCE?...57 576 5766 57666
IS THE NEXT TERM 576666?...YES
THE WINNING PROCEDURE WAS GROW-ONE FOR /DUMMY/ EQUALS 6
←OCCAM
SEQUENCE?...7Ø 61 52 43
IS THE NEXT TERM 34?...YES
THE WINNING PROCEDURE WAS DIM-ONE FOR /DUMMY/ EQUALS 9
←
```

The procedures SQUARES, MOD-FOUR, and NEXT-FACTORIAL can also be generalized in the same way, but the results are less useful. In all these cases, the generalized procedures simply describe subsets of the sequence described by the original, ungeneralized procedure. The original SQUARES, for example, describes the squares of the integers; the generalized SQUARES procedure is:

```
TO SQUARES /N/ /DUMMY/
1Ø MAKE "X" SUM OF SQUARE-ROOT OF /N/ AND /DUMMY/
2Ø OUTPUT PRODUCT OF /X/ AND /X/
END
```

Instead of describing sequences which are more general than the original, this procedure merely describes various subsets of the original sequence. Perhaps the most useful case is the subset consisting of the squares of the odd integers:

```
←OCCAM
SEQUENCE?...1 9 25 49
IS THE NEXT TERM 81?...YES
THE WINNING PROCEDURE WAS SQUARES FOR /DUMMY/ EQUALS 2
```

The procedures SQUARES, MOD-FOUR, and NEXT-FACTORIAL can, however, be otherwise defined in ways that lead to very useful generalizations. For example, if we define SQUARES:

```
TO SQUARES /N/
1Ø OUTPUT UP-ROOT OF 2 AND /N/
END
```

As before, SQUARES computes $(\sqrt{/N/}+1)^2$. It uses the procedure UP-ROOT which, in turn, uses two standard subprocedures, ROOT /M/ /N/ and POWER /M/ /N/, for getting /M/th roots and /M/th powers.

```
TO UP-ROOT /M/ AND /N/
1Ø MAKE "X" (ROOT OF /M/ AND /N/)  (This gets the /M/th root of /N/)
2Ø OUTPUT POWER OF /M/ AND          (This raises /X/+1 to the /M/th
    (SUM OF /X/ AND 1)               power)
END
```

When this version of SQUARES is generalized, it becomes:

```
TO SQUARES /N/ AND /DUMMY/
1Ø OUTPUT UP-ROOT OF /DUMMY/ AND /N/
END
```

And this is a genuinely useful extension -- it effectively generalizes SQUARES to describe sequences of integers raised to various integer powers. For example, if /DUMMY/ is 3, we get the sequence of cubes 1 8 27 64...; if /DUMMY/ is 4 we get the 4th powers 1 16 81 256... and so on.

Similarly, let us rewrite the MOD-FOUR successor as,

```
TO MOD-FOUR /N/
1Ø TEST IS /N/ 4
2Ø IF TRUE OUTPUT Ø
3Ø OUTPUT SUM OF /N/ AND 1
END
```

It will then truly generalize as the successor for the family of modulo-/DUMMY/ sequences (except when /N/ is greater than /DUMMY/).

```
TO MOD-FOUR /N/ /DUMMY/
1Ø TEST IS /N/ /DUMMY/
2Ø IF TRUE OUTPUT Ø
3Ø OUTPUT SUM OF /N/ AND 1
END
```

Thus, by exercising some care in how the original, ungeneralized procedures are defined, the strategy of generalizing procedures can have a wide scope of application. It can, however, lead to trouble with certain procedures. Consider, for example, what it does with the procedure DIV-TWO:

```
←OCCAM
SEQUENCE?...16 8 4
CAN'T DO THAT ONE.  TELL ME HOW AND THEN TYPE 'CONTINUE'
←TO DIV-TWO /N/
>1Ø OUTPUT QUOTIENT OF /N/ AND 2
>END
DIV-TWO DEFINED
←CON
PROCEDURE NAME?...DIV-TWO
←OCCAM
SEQUENCE?...8 4 2
DIVISION BY ZERO                    (Error comment - illegal operation)
I WAS AT LINE 1Ø IN DIV-TWO         (LOGO terminates the program)
←
```

The "bug" is easily located. When GENERAL-TESTER tried to test the generalized DIV-TWO procedure with its first /DUMMY/ value, Ø, it called DIV-TWO OF 8 AND Ø which attempted to divide 8 by Ø. We can fix this by writing DIV-TWO in a tricky way replacing the divisor 2, by SUM OF 1 AND 1. Thus, in the generalized form, /DUMMY/ will not replace the entire divisor, so division by zero will not occur.

```
←TO DIV-TWO /N/
>1Ø OUTPUT QUOTIENT OF /N/ AND (SUM OF 1 AND 1)
>END
DIV-TWO DEFINED
←CON
```

```
PROCEDURE NAME?...DIV-TWO
←OCCAM
SEQUENCE?...27 9 3
IS THE NEXT TERM 1?...YES
THE WINNING PROCEDURE WAS DIV-TWO FOR /DUMMY/ EQUALS 2
←LIST DIV-TWO
TO DIV-TWO /N/ /DUMMY/
1Ø OUTPUT QUOTIENT OF /N/ AND (SUM OF /DUMMY/ AND 1)
END
←
```

Situations of this kind, which call for rewriting a procedure to eliminate an illegal operation, are unavoidable but they do not occur frequently.

2.8 Transforming Sequences

Generalization has measurably increased OCCAM's power. Before we introduced GENERALIZE, OCCAM had a procedure-bank containing ten ungeneralized procedures and thus could identify segments of ten infinite sequences. Once again OCCAM has ten procedures in its banks, one ungeneralized and nine in the /GENERAL-BANK/. But each generalized procedure actually describes 11 different successor procedures (corresponding to 11 different values of /DUMMY/). So OCCAM can identify, all together, any segment of 1ØØ different sequences.

We can, of course, considerably expand the number of procedures in OCCAM without changing its structure. Adding new rules will widen its scope at the rate of one sequence for every ungeneralized procedure and 11 (or more if we increase the range of /DUMMY/) for every new entry in /GENERAL-BANK/. We will certainly add new rules, but there is a very different direction for expansion that will be far more effective in boosting OCCAM's power. Up to now we have concentrated our effort on a strategy for generalizing sequence generation rules. Let's look now at

another kind of strategy, for transforming the given sequences directly. As we noted before, even testing /DUMMY/ between Ø and 1ØØØ, OCCAM would fail to find the next term of the sequence:

1 1ØØ2 2ØØ3 3ØØ4 ...

Subtracting every term in this sequence from its successor, however, yields a new sequence:

1ØØ1 1ØØ1 1ØØ1

which can be described by a simple, nongeneralized procedure:

```
TO CONSTANT /N/                    (This is clearly a better rule
1Ø OUTPUT /N/                       than the generalized FIVERS for
END                                 describing constant sequences)
```

Assuming CONSTANT has been entered in /PROCEDURE-BANK/, OCCAM, in its present form, can easily find the next term in the *difference sequence*, i.e., the sequence made up of the difference of successive terms in the given sequence:

```
←OCCAM
SEQUENCE?...1ØØ1 1ØØ1 1ØØ1
IS THE NEXT TERM 1ØØ1?...YES
THE WINNING PROCEDURE WAS CONSTANT
←
```

We can now hazard a successor for the given sequence itself. Our method is based upon a truth and a supposition. The truth is: each term in a sequence is expressible as the sum of its predecessor and the corresponding term in its difference sequence. Now our supposition: the next term will *probably* be the sum of the last term in the original sequence (3ØØ4 in the example) and the winning next term of the difference sequence (1ØØ1), that is, 4ØØ5. Since this looks like an effective strategy (after all, it seemed to work here, the first time we used it), let's put it into OCCAM. We'll modify OCCAM so that, if its existing strategies

are unsuccessful, it will build the difference sequence, try to solve it, and use the solutions it finds to compute possible next terms for the original problem sequence.

We'll first write a procedure DIFF-SEQUENCE whose input is a sequence and whose output is the corresponding difference sequence. It simply outputs a sentence of differences of successive terms in the input sequence.

```
TO DIFF-SEQUENCE /SEQUENCE/
1Ø TEST EMPTYP BUTFIRST OF        (Have we subtracted all pairs?)
    /SEQUENCE/                    (Yes, done)
2Ø IF TRUE OUTPUT /EMPTY/           (If not, put the difference of
3Ø OUTPUT SENTENCE OF DIFFERENCE OF the first two terms in a
    (SECOND OF /SEQUENCE/) AND          sentence and repeat the
    (FIRST OF /SEQUENCE/)               process on the reduced
    AND DIFF-SEQUENCE OF BUTFIRST       sequence)
    OF /SEQUENCE/
END
```

DIFF-SEQUENCE will, of course, transform arithmetic sequences into constant sequences.

```
←PRINT DIFF-SEQUENCE OF "1 1ØØ2 2ØØ3 3ØØ4"
1ØØ1 1ØØ1 1ØØ1
←PRINT DIFF-SEQUENCE OF "66 77 88 99"
11 11 11
←PRINT DIFF-SEQUENCE OF "6Ø 52 44 36 28"
-8 -8 -8 -8
←
```

The usefulness of DIFF-SEQUENCE does not stop here, though. Various other kinds of sequences also yield difference sequences that are easier to solve than the originals:

```
←PRINT DIFF-SEQUENCE OF "121 144 169 196"
23 25 27
←PRINT DIFF-SEQUENCE OF "1 12 123 1234"
11 111 1111
```

```
←PRINT DIFF-SEQUENCE OF "5 5 6 8 8 9"
Ø 1 2 Ø 1
←
```

We shall now use DIFF-SEQUENCE to make a difference sequence out of any sequence that the procedures in SCAN-P-LIST and SCAN-G-LIST fail to extrapolate. If this difference sequence is tractable, we will add its potential successor to the last term of the original sequence and submit the result to APPROVAL. If the extrapolation fails, we will try again with other potential successors of the difference sequence. Let's write a procedure SCAN-DIFF to do this. SCAN-DIFF differs from SCAN-P-LIST only in the additional input, the computation of /NEXT-TERM/, and the form of its winning procedure declaration.

```
TO SCAN-DIFF /SEQUENCE/ AND /DIFF-SEQUENCE/ AND /PROCEDURES/
1Ø TEST EMPTYP /PROCEDURES/
2Ø IF TRUE OUTPUT /EMPTY/
3Ø TEST TESTER OF /DIFF-SEQUENCE/ (Does the first procedure de-
    AND FIRST OF /PROCEDURES/      scribe the difference sequence?)
4Ø IF FALSE OUTPUT SCAN-DIFF OF   (If not, try the rest of the
    /SEQUENCE/ AND /DIFF-SEQUENCE/ bank)
    AND BUTFIRST OF /PROCEDURES/
5Ø MAKE "NEXT-TERM" SUM OF        (If so, the next term is the
    (LAST OF /SEQUENCE/) AND       last term in /SEQUENCE/ plus
    (EXECUTE OF FIRST OF           the next term of
     /PROCEDURES/ AND              /DIFF-SEQUENCE/)
    LAST OF /DIFF-SEQUENCE/)
6Ø TEST APPROVAL OF /NEXT-TERM/
7Ø IF TRUE PRINT SENTENCE OF
    "THE WINNING PROCEDURE WAS"
    AND SENTENCE OF FIRST OF
    /PROCEDURES/ AND "APPLIED TO  (Note the additional comment)
    THE DIFFERENCES OF THE TERMS"
8Ø IF TRUE OUTPUT FIRST OF
    /PROCEDURES/
9Ø OUTPUT SCAN-DIFF OF /SEQUENCE/
    AND /DIFF-SEQUENCE/ AND
    BUTFIRST OF /PROCEDURES/
END
```

Let's also write a related procedure SCAN-GEN-DIFF, for performing the same process, using generalized procedures.

```
TO SCAN-GEN-DIFF /SEQUENCE/ AND /DIFF-SEQUENCE/ AND /PROCEDURES/
1Ø TEST EMPTYP /PROCEDURES/
2Ø IF TRUE OUTPUT /EMPTY/
3Ø MAKE "WINNING-DUMMY" GENERAL-
    TESTER OF /DIFF-SEQUENCE/     (Test the first procedure against
    AND FIRST OF /PROCEDURES/      the difference sequence)
4Ø TEST EMPTYP /WINNING-DUMMY/
5Ø IF TRUE OUTPUT SCAN-GEN-DIFF   (If it fails, go on to the rest
    OF /SEQUENCE/ AND              of the list)
    /DIFF-SEQUENCE/ AND
    BUTFIRST OF /PROCEDURES/
6Ø MAKE "NEXT-TERM" SUM OF        (If it wins, make the next term
    (LAST OF /SEQUENCE/) AND       the sum of the last term in
    (EXECUTE-2 OF (FIRST OF        /SEQUENCE/ and the next term
    /PROCEDURES/) AND              of /DIFF-SEQUENCE/)
    (LAST OF /DIFF-SEQUENCE/)
    AND (/WINNING-DUMMY/)
7Ø TEST APPROVAL OF /NEXT-TERM/
8Ø IF TRUE PRINT SENTENCE OF
    "THE WINNING PROCEDURE WAS"
    AND SENTENCE OF FIRST OF
    /PROCEDURES/ AND SENTENCE OF
    /DUMMY/ AND "APPLIED TO THE
    DIFFERENCES OF THE TERMS"
9Ø IF TRUE OUTPUT FIRST OF /PROCEDURES/
1ØØ OUTPUT SCAN-GEN-DIFF OF /SEQUENCE/
    AND /DIFF-SEQUENCE/ AND BUTFIRST
    OF /PROCEDURES/
END
```

To incorporate SCAN-DIFF and SCAN-GEN-DIFF in the extrapolation program, we need only add them as alternative tests in OCCAM. Lines 3Ø through 4Ø of OCCAM will now look like this:

```
3Ø TEST EMPTYP OF SCAN-P-LIST OF /SEQUENCE/ AND /PROCEDURE-BANK/
35 IF TRUE TEST EMPTYP OF SCAN-G-LIST OF /SEQUENCE/AND /GENERAL-
    BANK/
37 IF TRUE TEST EMPTYP OF SCAN-DIFF OF /SEQUENCE/ AND (DIFF-
    SEQUENCE OF /SEQUENCE/) AND /PROCEDURE-BANK/
39 IF TRUE TEST EMPTYP OF SCAN-GEN-DIFF OF /SEQUENCE/ AND
    (DIFF-SEQUENCE OF /SEQUENCE/) AND /GENERAL/BANK/
4Ø IF FALSE STOP
```

OCCAM is once again ready for testing. Let's start with our old problem:

```
←OCCAM
SEQUENCE?...1 1ØØ2 2ØØ3 3ØØ4
IS THE NEXT TERM 4ØØ5?...YES
THE WINNING PROCEDURE WAS CONSTANT APPLIED TO THE
DIFFERENCES OF THE TERMS
←
```

Let's show how OCCAM uses the differencing strategy to transform a variety of different kinds of sequences into tractable form:

```
←OCCAM
SEQUENCE?...75 6Ø 45 3Ø
IS THE NEXT TERM 15?...YES
THE WINNING PROCEDURE WAS CONSTANT APPLIED TO THE
DIFFERENCES OF THE TERMS

←OCCAM
SEQUENCE?...2 4 8
IS THE NEXT TERM 16?...NO
IS THE NEXT TERM 24?...YES
THE WINNING PROCEDURE WAS SQUARE APPLIED TO THE
DIFFERENCES OF THE TERMS
```

Frequently OCCAM will apply the strategy of differencing usefully to generalized procedures:

```
←OCCAM
SEQUENCE?...2 5 1Ø 17 26
IS THE NEXT TERM 37?...YES
THE WINNING PROCEDURE WAS ADD-TWO FOR /DUMMY/ EQUALS 2
APPLIED TO THE DIFFERENCES OF THE TERMS

←OCCAM
SEQUENCE?...1 12 123 1234
IS THE NEXT TERM 12345?...YES
THE WINNING PROCEDURE WAS GROW-ONE FOR /DUMMY/ EQUALS 1
APPLIED TO THE DIFFERENCES OF THE TERMS
←
```

OCCAM can now make a guess at the successor of all sequences whose difference sequences can be described by any of the procedures in the banks. Thus, it can solve such sequences as these:

```
←OCCAM
SEQUENCE?...1 1 2 2 3 3 4
IS THE NEXT TERM 4?...YES
THE WINNING PROCEDURE WAS MOD-FOUR FOR /DUMMY/ EQUALS 1
APPLIED TO THE DIFFERENCES OF THE TERMS

←OCCAM
SEQUENCE?...5Ø 66 74 78
IS THE NEXT TERM 8Ø?...YES
THE WINNING PROCEDURE WAS DIV-TWO FOR /DUMMY/ EQUALS 1
APPLIED TO THE DIFFERENCES OF THE TERMS

←OCCAM
SEQUENCE?...Ø 1 3 9 33
IS THE NEXT TERM 153?...YES
THE WINNING PROCEDURE WAS NEXT-FACTORIAL FOR /DUMMY/ EQUALS 1
APPLIED TO THE DIFFERENCES OF THE TERMS

←OCCAM
SEQUENCE?...1 2 6 15 31
IS THE NEXT TERM 56?...YES
THE WINNING PROCEDURE WAS SQUARES FOR /DUMMY/ EQUALS 2
APPLIED TO THE DIFFERENCES OF THE TERMS
```

OCCAM can now solve difficult and esoteric sequences as well as familiar, "natural" types; a high-school student is not likely to ask for a successor to the sequence 18 20 44 724..., for example, though OCCAM can give one. Also, the existing (generalizing and difference-transforming) strategies using the relatively small set of procedures presently in the banks, will often yield a large number of plausible extrapolations. Thus, for example, OCCAM can now extrapolate over a thousand possible successors for each sequence made up of two random numbers (though it is by no means assured of getting the "right" one).

The strategy of transforming sequences by taking differences of successive terms is one of a set of such effective strategies. Others of this kind include transforming a given sequence into the sequence comprised of its alternate terms, its partial sums (products, quotients) over one or more preceding terms, etc. Problems to incorporate these strategies are given in the associated problem sequel.

As the extrapolation program becomes larger both in the number of procedures it has explicitly to consider and the number of different strategies it has for modifying procedures or sequences, it can become unwieldy or inefficient. Under these circumstances, the use of higher order strategies or of administrative procedures, to improve the operation of OCCAM, becomes important. Some ideas for designing these kinds of facilities are also considered in the problem set.

Thus far, OCCAM has been developed to work in the single domain of sequences of *numbers*. With straightforward modification, it can be adapted for use with *letter sequences* or as the basis of a simple *function-guessing* situation. Further, using the same methods already built into OCCAM, the "inverse" program can be built for *generating* various kinds of sequences for the *user* to extrapolate. These extensions also are considered in the problem set.

Rather than studying further problem-solving strategies in the context of sequence guessing, we shall next study new kinds of strategies in yet another, distinctly different problem-solving situation, traversing a maze.

3. MAZES

3.1 Introduction - A Minimal History of Mazes

The most famous maze of antiquity, the Labyrinth, was a gigantic walled structure reputedly on the island of Crete designed by a man named Daedalus. It had two openings to the outside world between which lay not only an incredibly complex series of corridors, loops, blind alleys, and traps, but the "man-eating" Minotaur as well. Only three men are said to have escaped from the Labyrinth. Resourceful Daedalus (who, it seems, couldn't otherwise master his own creation) and his son Icarus, flew out the top with wings fashioned from wax and bird feathers; the clever hero Theseus trailed a thread behind him and, after he killed the Minotaur, he followed the thread back to the entrance and freedom.

After the Cretan empire fell and the Labyrinth was destroyed, real mazes fell somewhat from favor. They flourished once again, however, in 17th, 18th, and 19th century England, in palace gardens, constructed as carefully clipped barriers of impenetrable shrubbery. The plan of the maze at Hampton Court Palace, perhaps the most intricate of such mazes, is shown in Figure 1.

Most modern mazes are both more modest and less treacherous. Earlier in this century behavioral scientists found simple mazes useful for testing certain learning patterns with various animals. Unlike the English or Cretan varieties, these mazes generally had no exits -- the mouse or rat was simply dropped into the maze at some starting place and lifted out when it found the food.

Our task in the following pages will be to put ourselves in the place of the Minotaur's prey, or a bewildered Englishman, or a